MPI Assignment

Joseba Hernández Bravo and Jorge Vicente Puig

January 2022

Paralleling finite differences method

In this section we will explain the strategy we have followed for the parallelization of the sequential program *finite_differences*. We can distinguish three main parts to parallelize in our program:

- *Initial loop:* where we fill the matrices with the initial conditions.
- *Main loop:* where we iterate over the time filling the rest of the matrix.
- *Checksum Loop:* where we calculate the checksum for the final state.

The beginning of the parallelization consist on opening an MPI environment with `MPI_Init()` and `MPI_Finalize()` along with `MPI_Comm_size` and `MPI_Comm_rank` which returns the size of the communicator, i.e. the number of processors used, and a variable which identify the number of the processor used respectively.

Once the MPI environment is open, we proceed to create a `size` matrices `subU`, each one for each process. Each matrix will contain `n_rows+2` where `n_rows = X / size` and the proper initial conditions of the *finite differences* problem. Then, for first iteration of the numerical method we will make the computations for rows from 1 to `n_rows`. The problem arise when for iteration 2 each process need the updated rows that had computed another process.

Consequently, it is compulsory to make a message passing between different processors with the required data. For that purpose we have to keep in mind that there are three different cases:

- *Root case*, i.e. the case where `rank==0`: we will need the row `n_rows` for computations.
- *Last process*, i.e. the case where `rank==size-1`: we will need the row `0`.
- *Another process*, i.e. correspond to all the intermediate processes. This processes will take care of `n_rows`, we will need both first and last row.

For this communications, we have used the two functions `MPI_Recv()` and `MPI_Send()` as tools between the different processes for message passing.

Then, based on the case that we have to send/receive the data, that would be stored in the rows `0` and `n_rows+1`, the rows that were reserved from the beginning with this purpose. In the following image you can see how the algorithm works in a parallel way:

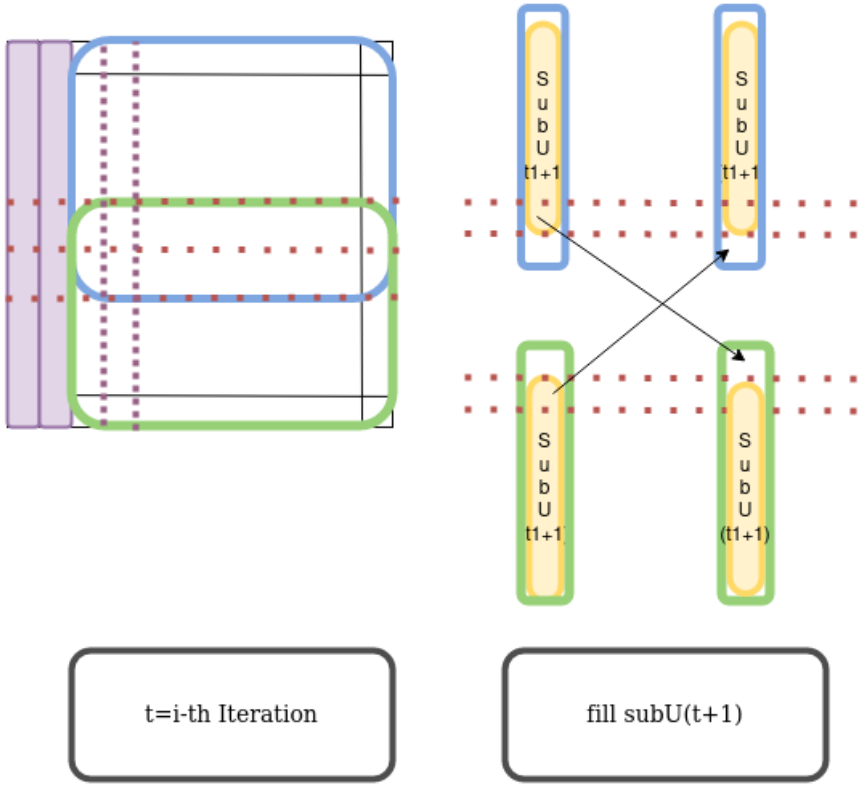

Figure 1: Image of how the algorithm works for time t=i for the special case of 2 processes. The purple boxes indicate the initial conditions, while the red dots indicates the separation between columns/rows.

Then, we calculate the checksum for the last column of each matrix and using the function `MPI_Reduce()` (with the argument `MPI_SUM`) we add up all the resulting values. Finally, we print the checksum on root process.

For the performance analysis we had make use of the `MPI_Wtime()`

Table 1: Performance analysis[1]

Nº processes	**X size**	**Elapsed time**
2	8000	1.626625
2	16000	5.917806
2	32000	20.245069
4	8000	1.543450
4	16000	4.975963
4	32000	17.902056
8	8000	1.237275
2	16000	2.966478
8	32000	14.175387

[1] It has been set `T` as 20.000.

Looking at the above table we can extract some conclusions:

- For each matrix size the elapsed time is reduced when increasing the number of processors.
- The importance of parallelization arise when increasing the problem size: for size 8000 there is not significant difference whereas for size 32000 it is obtained a reduction of elapse time of 6 seconds. More big is the problem size more importance gets the parallelization process.
- The ratio of reduction on execution time is not proportional with the increase on number of processors, we can duplicate the number of processes and not reducing elapsed time by 2.
- Due to only having the elapse time of the computations we do not have the proper measures to compare between different problem sizes.

Paralleling laplace method

For paralleling the `laplace.c` program we will follow a similar reasoning as the one described before.

Once the MPI environment is initialised, we will define two matrices: `A` and `A_new`. Each of these matrices will be initialised in every process, *i.e.*, if we run the program using 8 processes, we will create and fill the same number of matrices. In addition, the number of rows in each of these matrices will be exactly equal to the total number of rows divided by the number of processes plus two.

At this point, unlike in the previous programme, we need an **entire row** for each process in order to compute all the new values for the next iteration. That is why we have defined the number of rows as `(TOT_ROWS/size)+2`. For each new cycle (or iteration) we will fill all the matrices up to the penultimate row and then we will make use of `MPI_Send()` and `MPI_Recv()`, to send and receive the remaining rows.

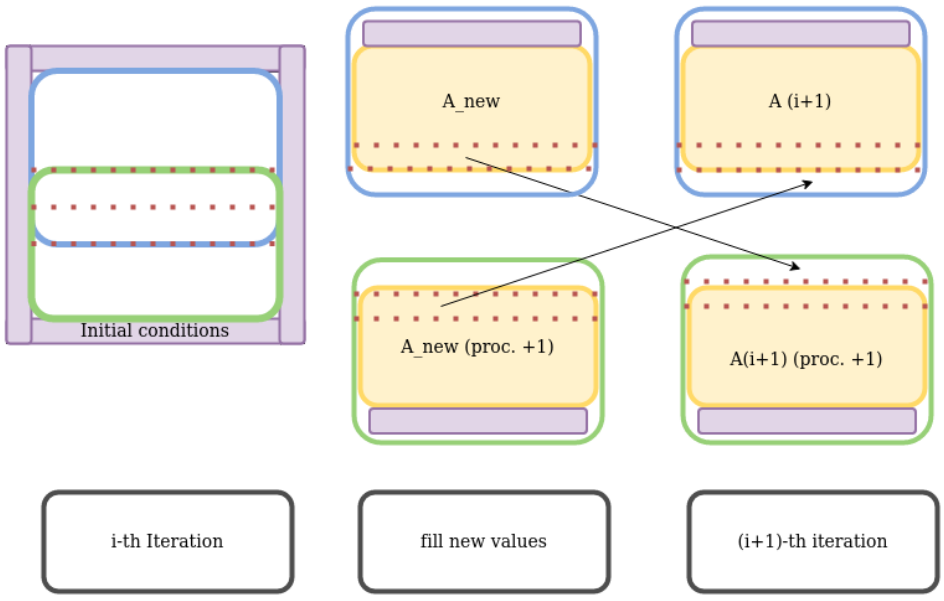

Figure 2: Image of how the algorithm works for the case of 2 processes.

Finally, we calculate for each matrices belonging to each process its corresponding error and then, by means of the function `MPI_Reduce()`, we calculate the total error.

As in the finite case, we have made use of the `MPI_Wtime()` function to calculate the execution times for different matrix sizes and for different numbers of processes.

Table 2: Performance analysis for `Laplace`[1]

Nº processes	**X size**	**Elapsed time**
2	800	3.883530
2	8000	93.280910
2	80000	Too slow[2]
4	800	3.2405346666
4	8000	34.966908
4	80000	300.965576
8	800	2.108546
8	8000	19.405604
8	80000	194.024341

[1] It has been set `T` as and `max_iter` as .

[2] It was cancelled by the cluster for taking lot of execution time

Looking at the above table we can extract some conclusions, similar at the ones obtained on *finite differences*:

- For each matrix size the elapsed time is reduced when increasing the number of processes.
- A parallelization strategy can be crucial for large problem sizes, we can go from impossibility of computation *(case np 2, X 80000)* to a reasonable execution time *(case np 8, X 80000)*.
- As in previous case, the ratio of reduction on execution time is not proportional with the increase on number of processors.

- Due to only having the elapse time of the computations we do not have the proper measures to compare between different problem sizes.